Object serialization

Java 1.1 has added an interesting feature called *object serialization* that allows you to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be restored fully into the original object. This is even true across a network, which means that the serialization mechanism automatically compensates for differences in operating systems. That is, you can create an object on a Windows machine, serialize it, and send it across the network to a Unix machine where it will be correctly reconstructed. You don't have to worry about the data representations on the different machines, the byte ordering, or any other details.

By itself, object serialization is interesting because it allows you to implement *lightweight persistence*. Remember that persistence means an object's lifetime is not determined by whether a program is executing – the object lives *in between* invocations of the program. By taking a serializable object and writing it to disk, then restoring that object when the program is re-invoked, you're able to produce the effect of persistence. The reason it's called "lightweight" is that you can't simply define an object using some kind of "persistent" keyword and let the system take care of the details (although this might happen in the future). Instead, you must explicitly serialize and de-serialize the objects in your program.

Object serialization was added to the language to support two major features. Java 1.1's *remote method invocation* (RMI) allows objects that live on other machines to behave as if they live on your machine. When sending messages to remote objects, object serialization is necessary to transport the arguments and return values. RMI is discussed in Chapter 15.

Object serialization is also necessary for Java Beans, introduced in Java 1.1. When a Bean is used, its state information is generally configured at design time. This state information must be stored and later recovered when the program is started; object serialization performs this task.

Serializing an object is quite simple, as long as the object implements the **Serializable** interface (this interface is just a flag and has no methods). In Java 1.1, many standard library classes have been changed so they're serializable, including all of the wrappers for the primitive types, all of the collection classes, and many others. Even **Class** objects can be serialized. (See Chapter 11 for the implications of this.)

To serialize an object, you create some sort of **OutputStream** object and then wrap it inside an **ObjectOutputStream** object. At this point you need only call **writeObject()** and your object is serialized and sent to the **OutputStream**. To reverse the process, you wrap an **InputStream** inside an **ObjectInputStream** and call **readObject()**. What comes back is, as usual, a handle to an upcast **Object**, so you must downcast to set things straight.

A particularly clever aspect of object serialization is that it not only saves an image of your object but it also follows all the handles contained in your object and saves *those* objects, and follows all the handles in each of those objects, etc. This is sometimes referred to as the "web of objects" that a single object can be connected to, and it includes arrays of handles to objects as well as member objects. If you had to maintain your own object serialization scheme, maintaining the code to follow all these links would be a bit mind-boggling. However, Java object serialization seems to pull it off flawlessly, no doubt using an optimized algorithm that traverses the web of objects. The following example tests the serialization mechanism by making a "worm" of linked objects, each of which has a link to the next segment in the worm as well as an array of handles to objects of a different class, **Data**:

```
//: Worm.java
// Demonstrates object serialization in Java 1.1
import java.io.*;

class Data implements Serializable {
  private int i;
  Data(int x) { i = x; }
  public String toString() {
    return Integer.toString(i);
  }
}

public class Worm implements Serializable {
  // Generate a random int value:
  private static int r() {
    return (int)(Math.random() * 10);
  }
  private Data[] d = {
    new Data(r()), new Data(r()), new Data(r())
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
```

```
Worm(int i, char x) {
  System.out.println(" Worm constructor: " + i);
  c = x;
  if(--i > 0)
    next = new Worm(i, (char)(x + 1));
}
Worm() {
  System.out.println("Default constructor");
}
public String toString() {
  String s = ":" + c + "(";
  for(int i = 0; i < d.length; i++)
    s += d[i].toString();
  s += ")";
  if(next != null)
    s += next.toString();
  return s;
}
public static void main(String[] args) {
  Worm w = new Worm(6, 'a');
  System.out.println("w = " + w);
  try {
    ObjectOutputStream out =
      new ObjectOutputStream(
        new FileOutputStream("worm.out"));
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.close(); // Also flushes output
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
    System.out.println(s + ", w2 = " + w2);
  } catch(Exception e) {
    e.printStackTrace();
  }
  try {
    ByteArrayOutputStream bout =
      new ByteArrayOutputStream();
    ObjectOutputStream out =
      new ObjectOutputStream(bout);
    out.writeObject("Worm storage");
    out.writeObject(w);
```

```
        out.flush();
        ObjectInputStream in =
          new ObjectInputStream(
            new ByteArrayInputStream(
              bout.toByteArray()));
        String s = (String)in.readObject();
        Worm w3 = (Worm)in.readObject();
        System.out.println(s + ", w3 = " + w3);
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
} ///:~
```

To make things interesting, the array of **Data** objects inside **Worm** are initialized with random numbers. (This way you don't suspect the compiler of keeping some kind of meta-information.) Each **Worm** segment is labeled with a **char** that's automatically generated in the process of recursively generating the linked list of **Worm**s. When you create a **Worm**, you tell the constructor how long you want it to be. To make the **next** handle it calls the **Worm** constructor with a length of one less, etc. The final **next** handle is left as **null**, indicating the end of the **Worm**.

The point of all this was to make something reasonably complex that couldn't easily be serialized. The act of serializing, however, is quite simple. Once the **ObjectOutputStream** is created from some other stream, **writeObject( )** serializes the object. Notice the call to **writeObject( )** for a **String**, as well. You can also write all the primitive data types using the same methods as **DataOutputStream** (they share the same interface).

There are two separate **try** blocks that look similar. The first writes and reads a file and the second, for variety, writes and reads a **ByteArray**. You can read and write an object using serialization to any **DataInputStream** or **DataOutputStream** including, as you will see in the networking chapter, a network. The output from one run was:

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
```

```
Worm storage, w2 =
:a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3 =
:a(262):b(100):c(396):d(480):e(316):f(398)
```

You can see that the deserialized object really does contain all of the links that were in the original object.

Note that no constructor, not even the default constructor, is called in the process of deserializing a **Serializable** object. The entire object is restored by recovering data from the **InputStream**.

Object serialization is another Java 1.1 feature that is not part of the new **Reader** and **Writer** hierarchies, but instead uses the old **InputStream** and **OutputStream** hierarchies. Thus you might encounter situations in which you're forced to mix the two hierarchies.

Finding the class

You might wonder what's necessary for an object to be recovered from its serialized state. For example, suppose you serialize an object and send it as a file or through a network to another machine. Could a program on the other machine reconstruct the object using only the contents of the file?

The best way to answer this question is (as usual) by performing an experiment. The following file goes in the subdirectory for this chapter:

```
//: Alien.java
// A serializable class
import java.io.*;

public class Alien implements Serializable {
} ///:~
```

The file that creates and serializes an **Alien** object goes in the same directory:

```
//: FreezeAlien.java
// Create a serialized output file
import java.io.*;

public class FreezeAlien {
  public static void main(String[] args)
      throws Exception {
    ObjectOutput out =
```

```
      new ObjectOutputStream(
        new FileOutputStream("file.x"));
    Alien zorcon = new Alien();
    out.writeObject(zorcon);
  }
} ///:~
```

Rather than catching and handling exceptions, this program takes the quick and dirty approach of passing the exceptions out of **main( )**, so they'll be reported on the command line.

Once the program is compiled and run, copy the resulting **file.x** to a subdirectory called **xfiles**, where the following code goes:

```
//: ThawAlien.java
// Try to recover a serialized file without the
// class of object that's stored in that file.
package c10.xfiles;
import java.io.*;

public class ThawAlien {
  public static void main(String[] args)
      throws Exception {
    ObjectInputStream in =
      new ObjectInputStream(
        new FileInputStream("file.x"));
    Object mystery = in.readObject();
    System.out.println(
      mystery.getClass().toString());
  }
} ///:~
```

This program opens the file and reads in the object **mystery** successfully. However, as soon as you try to find out anything about the object – which requires the **Class** object for **Alien** – the Java Virtual Machine (JVM) cannot find **Alien.class** (unless it happens to be in the Classpath, which it shouldn't be in this example). You'll get a **ClassNotFoundException.** (Once again, all evidence of alien life vanishes before proof of its existence can be verified!)

If you expect to do much after you've recovered an object that has been serialized, you must make sure that the JVM can find the associated **.class** file either in the local class path or somewhere on the Internet.

Controlling serialization

As you can see, the default serialization mechanism is trivial to use. But what if you have special needs? Perhaps you have special security issues and you don't want to serialize portions of your object, or perhaps it just doesn't make sense for one sub-object to be serialized if that part needs to be created anew when the object is recovered.

You can control the process of serialization by implementing the **Externalizable** interface instead of the **Serializable** interface. The **Externalizable** interface extends the **Serializable** interface and adds two methods, **writeExternal( )** and **readExternal( )**, that are automatically called for your object during serialization and deserialization so that you can perform your special operations.

The following example shows simple implementations of the **Externalizable** interface methods. Note that **Blip1** and **Blip2** are nearly identical except for a subtle difference (see if you can discover it by looking at the code):

```
//: Blips.java
// Simple use of Externalizable & a pitfall
import java.io.*;
import java.util.*;

class Blip1 implements Externalizable {
  public Blip1() {
    System.out.println("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip1.readExternal");
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    System.out.println("Blip2 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
```

```
    System.out.println("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
    System.out.println("Blip2.readExternal");
  }
}

public class Blips {
  public static void main(String[] args) {
    System.out.println("Constructing objects:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Blips.out"));
      System.out.println("Saving objects:");
      o.writeObject(b1);
      o.writeObject(b2);
      o.close();
      // Now get them back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Blips.out"));
      System.out.println("Recovering b1:");
      b1 = (Blip1)in.readObject();
      // OOPS! Throws an exception:
//!   System.out.println("Recovering b2:");
//!   b2 = (Blip2)in.readObject();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The output for this program is:

```
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
```

```
Blip1 Constructor
Blip1.readExternal
```

The reason that the **Blip2** object is not recovered is that trying to do so causes an exception. Can you see the difference between **Blip1** and **Blip2**? The constructor for **Blip1** is **public**, while the constructor for **Blip2** is not, and that causes the exception upon recovery. Try making **Blip2**'s constructor **public** and removing the //! comments to see the correct results.

When **b1** is recovered, the **Blip1** default constructor is called. This is different from recovering a **Serializable** object, in which the object is constructed entirely from its stored bits, with no constructor calls. With an **Externalizable** object, all the normal default construction behavior occurs (including the initializations at the point of field definition), and *then* **readExternal( )** is called. You need to be aware of this – in particular the fact that all the default construction always takes place – to produce the correct behavior in your **Externalizable** objects.

Here's an example that shows what you must do to fully store and retrieve an **Externalizable** object:

```
//: Blip3.java
// Reconstructing an externalizable object
import java.io.*;
import java.util.*;

class Blip3 implements Externalizable {
  int i;
  String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialized only in non-default
    // constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip3.writeExternal");
```

```
    // You must do this:
    out.writeObject(s); out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // You must do this:
    s = (String)in.readObject();
    i =in.readInt();
  }
  public static void main(String[] args) {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3.toString());
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Blip3.out"));
      System.out.println("Saving object:");
      o.writeObject(b3);
      o.close();
      // Now get it back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Blip3.out"));
      System.out.println("Recovering b3:");
      b3 = (Blip3)in.readObject();
      System.out.println(b3.toString());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The fields **s** and **i** are initialized only in the second constructor, but not in the default constructor. This means that if you don't initialize **s** and **i** in **readExternal**, it will be **null** (since the storage for the object gets wiped to zero in the first step of object creation). If you comment out the two lines of code following the phrases "You must do this" and run the program, you'll see that when the object is recovered, **s** is **null** and **i** is zero.

If you are inheriting from an **Externalizable** object, you'll typically call the base-class versions of **writeExternal( )** and **readExternal( )** to provide proper storage and retrieval of the base-class components.

So to make things work correctly you must not only write the important data from the object during the **writeExternal( )** method (there is no default behavior that writes any of the member objects for an **Externalizable** object), but you must also recover that data in the **readExternal( )** method. This can be a bit confusing at first because the default construction behavior for an **Externalizable** object can make it seem like some kind of storage and retrieval takes place automatically. It does not.

The transient keyword

When you're controlling serialization, there might be a particular subobject that you don't want Java's serialization mechanism to automatically save and restore. This is commonly the case if that subobject represents sensitive information that you don't want to serialize, such as a password. Even if that information is **private** in the object, once it's serialized it's possible for someone to access it by reading a file or intercepting a network transmission.

One way to prevent sensitive parts of your object from being serialized is to implement your class as **Externalizable**, as shown previously. Then nothing is automatically serialized and you can explicitly serialize only the necessary parts inside **writeExternal( )**.

If you're working with a **Serializable** object, however, all serialization happens automatically. To control this, you can turn off serialization on a field-by-field basis using the **transient** keyword, which says "Don't bother saving or restoring this – I'll take care of it."

For example, consider a **Login** object that keeps information about a particular login session. Suppose that, once you verify the login, you want to store the data, but without the password. The easiest way to do this is by implementing **Serializable** and marking the **password** field as **transient**. Here's what it looks like:

```
//: Logon.java
// Demonstrates the "transient" keyword
import java.io.*;
import java.util.*;

class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  Logon(String name, String pwd) {
```

```
    username = name;
    password = pwd;
  }
  public String toString() {
    String pwd =
      (password == null) ? "(n/a)" : password;
    return "logon info: \n   " +
      "username: " + username +
      "\n   date: " + date.toString() +
      "\n   password: " + pwd;
  }
  public static void main(String[] args) {
    Logon a = new Logon("Hulk", "myLittlePony");
    System.out.println( "logon a = " + a);
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Logon.out"));
      o.writeObject(a);
      o.close();
      // Delay:
      int seconds = 5;
      long t = System.currentTimeMillis()
          + seconds * 1000;
      while(System.currentTimeMillis() < t)
        ;
      // Now get them back:
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("Logon.out"));
      System.out.println(
        "Recovering object at " + new Date());
      a = (Logon)in.readObject();
      System.out.println( "logon a = " + a);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

You can see that the **date** and **username** fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, and so is not stored to disk; also the serialization mechanism makes no attempt to recover it. The output is:

```
logon a = logon info:
   username: Hulk
   date: Sun Mar 23 18:25:53 PST 1997
   password: myLittlePony
Recovering object at Sun Mar 23 18:25:59 PST 1997
logon a = logon info:
   username: Hulk
   date: Sun Mar 23 18:25:53 PST 1997
   password: (n/a)
```

When the object is recovered, the **password** field is **null**. Note that **toString( )** must check for a **null** value of **password** because if you try to assemble a **String** object using the overloaded '**+**' operator, and that operator encounters a **null** handle, you'll get a **NullPointerException**. (Newer versions of Java might contain code to avoid this problem.)

You can also see that the **date** field is stored to and recovered from disk and not generated anew.

Since **Externalizable** objects do not store any of their fields by default, the **transient** keyword is for use with **Serializable** objects only.

An alternative to Externalizable

If you're not keen on implementing the **Externalizable** interface, there's another approach. You can implement the **Serializable** interface and *add* (notice I say "add" and not "override" or "implement") methods called **writeObject( )** and **readObject( )** that will automatically be called when the object is serialized and deserialized, respectively. That is, if you provide these two methods they will be used instead of the default serialization.

The methods must have these exact signatures:

```
private void
  writeObject(ObjectOutputStream stream)
    throws IOException;

private void
  readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

From a design standpoint, things get really weird here. First of all, you might think that because these methods are not part of a base class or the **Serializable** interface, they ought to be defined in their own interface(s). But notice that they are defined as **private**, which means they are to be

called only by other members of this class. However, you don't actually call them from other members of this class, but instead the **writeObject( )** and **readObject( )** methods of the **ObjectOutputStream** and **ObjectInputStream** objects call your object's **writeObject( )** and **readObject( )** methods. (Notice my tremendous restraint in not launching into a long diatribe about using the same method names here. In a word: confusing.) You might wonder how the **ObjectOutputStream** and **ObjectInputStream** objects have access to **private** methods of your class. We can only assume that this is part of the serialization magic.

In any event, anything defined in an **interface** is automatically **public** so if **writeObject( )** and **readObject( )** must be **private**, then they can't be part of an **interface**. Since you must follow the signatures exactly, the effect is the same as if you're implementing an **interface**.

It would appear that when you call **ObjectOutputStream.writeObject( )**, the **Serializable** object that you pass it to is interrogated (using reflection, no doubt) to see if it implements its own **writeObject( )**. If so, the normal serialization process is skipped and the **writeObject( )** is called. The same sort of situation exists for **readObject( )**.

There's one other twist. Inside your **writeObject( )**, you can choose to perform the default **writeObject( )** action by calling **defaultWriteObject( )**. Likewise, inside **readObject( )** you can call **defaultReadObject( )**. Here is a simple example that demonstrates how you can control the storage and retrieval of a **Serializable** object:

```
//: SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import java.io.*;

public class SerialCtl implements Serializable {
  String a;
  transient String b;
  public SerialCtl(String aa, String bb) {
    a = "Not Transient: " + aa;
    b = "Transient: " + bb;
  }
  public String toString() {
    return a + "\n" + b;
  }
  private void
    writeObject(ObjectOutputStream stream)
```

```
    throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void
    readObject(ObjectInputStream stream)
      throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String)stream.readObject();
  }
  public static void main(String[] args) {
    SerialCtl sc =
      new SerialCtl("Test1", "Test2");
    System.out.println("Before:\n" + sc);
    ByteArrayOutputStream buf =
      new ByteArrayOutputStream();
    try {
      ObjectOutputStream o =
        new ObjectOutputStream(buf);
      o.writeObject(sc);
      // Now get it back:
      ObjectInputStream in =
        new ObjectInputStream(
          new ByteArrayInputStream(
            buf.toByteArray()));
      SerialCtl sc2 = (SerialCtl)in.readObject();
      System.out.println("After:\n" + sc2);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

In this example, one **String** field is ordinary and the other is **transient**, to prove that the non-**transient** field is saved by the **defaultWriteObject( )** method and the **transient** field is saved and restored explicitly. The fields are initialized inside the constructor rather than at the point of definition to prove that they are not being initialized by some automatic mechanism during deserialization.

If you are going to use the default mechanism to write the non-**transient** parts of your object, you must call **defaultWriteObject( )** as the first operation in **writeObject( )** and **defaultReadObject( )** as the first operation in **readObject( )**. These are strange method calls. It would appear, for example, that you are calling **defaultWriteObject( )** for an

ObjectOutputStream and passing it no arguments, and yet it somehow turns around and knows the handle to your object and how to write all the non-**transient** parts. Spooky.

The storage and retrieval of the **transient** objects uses more familiar code. And yet, think about what happens here. In **main( )**, a **SerialCtl** object is created, and then it's serialized to an **ObjectOutputStream.** (Notice in this case that a buffer is used instead of a file – it's all the same to the **ObjectOutputStream.**) The serialization occurs in the line:

```
o.writeObject(sc);
```

The **writeObject( )** method must be examining **sc** to see if it has its own **writeObject( )** method. (Not by checking the interface – there isn't one – or the class type, but by actually hunting for the method using reflection.) If it does, it uses that. A similar approach holds true for **readObject( )**. Perhaps this was the only practical way that they could solve the problem, but it's certainly strange.

Versioning

It's possible that you might want to change the version of a serializable class (objects of the original class might be stored in a database, for example). This is supported but you'll probably do it only in special cases, and it requires an extra depth of understanding that we will not attempt to achieve here. The JDK1.1 HTML documents downloadable from Sun (which might be part of your Java package's online documents) cover this topic quite thoroughly.

Using persistence

It's quite appealing to use serialization technology to store some of the state of your program so that you can easily restore the program to the current state later. But before you can do this, some questions must be answered. What happens if you serialize two objects that both have a handle to a third object? When you restore those two objects from their serialized state, do you get only one occurrence of the third object? What if you serialize your two objects to separate files and deserialize them in different parts of your code?

Here's an example that shows the problem:

```
//: MyWorld.java
import java.io.*;
import java.util.*;
```

```
class House implements Serializable {}

class Animal implements Serializable {
  String name;
  House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
  }
  public String toString() {
    return name + "[" + super.toString() +
      "], " + preferredHouse + "\n";
  }
}

public class MyWorld {
  public static void main(String[] args) {
    House house = new House();
    Vector  animals = new Vector();
    animals.addElement(
      new Animal("Bosco the dog", house));
    animals.addElement(
      new Animal("Ralph the hamster", house));
    animals.addElement(
      new Animal("Fronk the cat", house));
    System.out.println("animals: " + animals);

    try {
      ByteArrayOutputStream buf1 =
        new ByteArrayOutputStream();
      ObjectOutputStream o1 =
        new ObjectOutputStream(buf1);
      o1.writeObject(animals);
      o1.writeObject(animals); // Write a 2nd set
      // Write to a different stream:
      ByteArrayOutputStream buf2 =
        new ByteArrayOutputStream();
      ObjectOutputStream o2 =
        new ObjectOutputStream(buf2);
      o2.writeObject(animals);
      // Now get them back:
      ObjectInputStream in1 =
        new ObjectInputStream(
```

```
            new ByteArrayInputStream(
              buf1.toByteArray()));
        ObjectInputStream in2 =
          new ObjectInputStream(
            new ByteArrayInputStream(
              buf2.toByteArray()));
        Vector animals1 = (Vector)in1.readObject();
        Vector animals2 = (Vector)in1.readObject();
        Vector animals3 = (Vector)in2.readObject();
        System.out.println("animals1: " + animals1);
        System.out.println("animals2: " + animals2);
        System.out.println("animals3: " + animals3);
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
  } ///:~
```

One thing that's interesting here is that it's possible to use object serialization to and from a byte array as a way of doing a "deep copy" of any object that's **Serializable.** (A deep copy means that you're duplicating the entire web of objects, rather than just the basic object and its handles.) Copying is covered in depth in Chapter 12.

Animal objects contain fields of type **House**. In **main( )**, a **Vector** of these **Animals** is created and it is serialized twice to one stream and then again to a separate stream. When these are deserialized and printed, you see the following results for one run (the objects will be in different memory locations each run):

```
animals: [Bosco the dog[Animal@1cc76c],
House@1cc769
, Ralph the hamster[Animal@1cc76d], House@1cc769
, Fronk the cat[Animal@1cc76e], House@1cc769
]
animals1: [Bosco the dog[Animal@1cca0c],
House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals2: [Bosco the dog[Animal@1cca0c],
House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
```

```
animals3: [Bosco the dog[Animal@1cca52],
House@1cca5c
, Ralph the hamster[Animal@1cca5d], House@1cca5c
, Fronk the cat[Animal@1cca61], House@1cca5c
]
```

Of course you expect that the deserialized objects have different addresses from their originals. But notice that in **animals1** and **animals2** the same addresses appear, including the references to the **House** object that both share. On the other hand, when **animals3** is recovered the system has no way of knowing that the objects in this other stream are aliases of the objects in the first stream, so it makes a completely different web of objects.

As long as you're serializing everything to a single stream, you'll be able to recover the same web of objects that you wrote, with no accidental duplication of objects. Of course, you can change the state of your objects in between the time you write the first and the last, but that's your responsibility – the objects will be written in whatever state they are in (and with whatever connections they have to other objects) at the time you serialize them.

The safest thing to do if you want to save the state of a system is to serialize as an "atomic" operation. If you serialize some things, do some other work, and serialize some more, etc., then you will not be storing the system safely. Instead, put all the objects that comprise the state of your system in a single collection and simply write that collection out in one operation. Then you can restore it with a single method call as well.

The following example is an imaginary computer-aided design (CAD) system that demonstrates the approach. In addition, it throws in the issue of **static** fields – if you look at the documentation you'll see that **Class** is **Serializable**, so it should be easy to store the **static** fields by simply serializing the **Class** object. That seems like a sensible approach, anyway.

```
//: CADState.java
// Saving and restoring the state of a
// pretend CAD system.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
  public static final int
    RED = 1, BLUE = 2, GREEN = 3;
  private int xPos, yPos, dimension;
```

```
  private static Random r = new Random();
  private static int counter = 0;
  abstract public void setColor(int newColor);
  abstract public int getColor();
  public Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass().toString() +
      " color[" + getColor() +
      "] xPos[" + xPos +
      "] yPos[" + yPos +
      "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.nextInt() % 100;
    switch(counter++ % 3) {
      default:
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static int color = RED;
  public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) {
    color = newColor;
  }
  public int getColor() {
    return color;
  }
}

class Square extends Shape {
  private static int color;
```

```
    public Square(int xVal, int yVal, int dim) {
      super(xVal, yVal, dim);
      color = RED;
    }
    public void setColor(int newColor) {
      color = newColor;
    }
    public int getColor() {
      return color;
    }
  }

  class Line extends Shape {
    private static int color = RED;
    public static void
    serializeStaticState(ObjectOutputStream os)
        throws IOException {
      os.writeInt(color);
    }
    public static void
    deserializeStaticState(ObjectInputStream os)
        throws IOException {
      color = os.readInt();
    }
    public Line(int xVal, int yVal, int dim) {
      super(xVal, yVal, dim);
    }
    public void setColor(int newColor) {
      color = newColor;
    }
    public int getColor() {
      return color;
    }
  }

  public class CADState {
    public static void main(String[] args)
        throws Exception {
      Vector shapeTypes, shapes;
      if(args.length == 0) {
        shapeTypes = new Vector();
        shapes = new Vector();
        // Add handles to the class objects:
        shapeTypes.addElement(Circle.class);
```

```
        shapeTypes.addElement(Square.class);
        shapeTypes.addElement(Line.class);
        // Make some shapes:
        for(int i = 0; i < 10; i++)
          shapes.addElement(Shape.randomFactory());
        // Set all the static colors to GREEN:
        for(int i = 0; i < 10; i++)
          ((Shape)shapes.elementAt(i))
            .setColor(Shape.GREEN);
        // Save the state vector:
        ObjectOutputStream out =
          new ObjectOutputStream(
            new FileOutputStream("CADState.out"));
        out.writeObject(shapeTypes);
        Line.serializeStaticState(out);
        out.writeObject(shapes);
      } else { // There's a command-line argument
        ObjectInputStream in =
          new ObjectInputStream(
            new FileInputStream(args[0]));
        // Read in the same order they were written:
        shapeTypes = (Vector)in.readObject();
        Line.deserializeStaticState(in);
        shapes = (Vector)in.readObject();
      }
      // Display the shapes:
      System.out.println(shapes);
    }
} ///:~
```

The **Shape** class **implements Serializable**, so anything that is inherited from **Shape** is automatically **Serializable** as well. Each **Shape** contains data, and each derived **Shape** class contains a **static** field that determines the color of all of those types of **Shapes**. (Placing a **static** field in the base class would result in only one field, since **static** fields are not duplicated in derived classes.) Methods in the base class can be overridden to set the color for the various types (**static** methods are not dynamically bound, so these are normal methods). The **randomFactory( )** method creates a different **Shape** each time you call it, using random values for the **Shape** data.

Circle and **Square** are straightforward extensions of **Shape**; the only difference is that **Circle** initializes **color** at the point of definition and **Square** initializes it in the constructor. We'll leave the discussion of **Line** for later.

In **main( )**, one **Vector** is used to hold the **Class** objects and the other to hold the shapes. If you don't provide a command line argument the **shapeTypes Vector** is created and the **Class** objects are added, and then the **shapes Vector** is created and **Shape** objects are added. Next, all the **static color** values are set to **GREEN**, and everything is serialized to the file **CADState.out**.

If you provide a command line argument (presumably **CADState.out**), that file is opened and used to restore the state of the program. In both situations, the resulting **Vector** of **Shapes** is printed out. The results from one run are:

```
>java CADState
[class Circle color[3] xPos[-51] yPos[-99] dim[38]
, class Square color[3] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[3] xPos[-70] yPos[1] dim[16]
, class Square color[3] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[3] xPos[-75] yPos[-43]
dim[22]
, class Square color[3] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[3] xPos[17] yPos[90] dim[-76]
]

>java CADState CADState.out
[class Circle color[1] xPos[-51] yPos[-99] dim[38]
, class Square color[0] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[1] xPos[-70] yPos[1] dim[16]
, class Square color[0] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[1] xPos[-75] yPos[-43]
dim[22]
, class Square color[0] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[1] xPos[17] yPos[90] dim[-76]
]
```

You can see that the values of **xPos**, **yPos**, and **dim** were all stored and recovered successfully, but there's something wrong with the retrieval of the **static** information. It's all '3' going in, but it doesn't come out that way. **Circles** have a value of 1 (**RED**, which is the definition), and **Squares** have a value of 0 (remember, they are initialized in the

constructor). It's as if the **statics** didn't get serialized at all! That's right – even though class **Class** is **Serializable**, it doesn't do what you expect. So if you want to serialize **statics**, you must do it yourself.

This is what the **serializeStaticState( )** and **deserializeStaticState( )** **static** methods in **Line** are for. You can see that they are explicitly called as part of the storage and retrieval process. (Note that the order of writing to the serialize file and reading back from it must be maintained.) Thus to make **CADState.java** run correctly you must (1) Add a **serializeStaticState( )** and **deserializeStaticState( )** to the shapes, (2) Remove the **Vector shapeTypes** and all code related to it, and (3) Add calls to the new serialize and deserialize static methods in the shapes.

Another issue you might have to think about is security, since serialization also saves **private** data. If you have a security issue, those fields should be marked as **transient**. But then you have to design a secure way to store that information so that when you do a restore you can reset those **private** variables.

Summary

The Java IO stream library does seem to satisfy the basic requirements: you can perform reading and writing with the console, a file, a block of memory, or even across the Internet (as you will see in Chapter 15). It's possible (by inheriting from **InputStream** and **OutputStream**) to create new types of input and output objects. And you can even add a simple extensibility to the kinds of objects a stream will accept by redefining the **toString( )** method that's automatically called when you pass an object to a method that's expecting a **String** (Java's limited "automatic type conversion").

There are questions left unanswered by the documentation and design of the IO stream library. For example, it would have been nice if you could say that you want an exception thrown if you try to overwrite a file when opening it for output – some programming systems allow you to specify that you want to open an output file, but only if it doesn't already exist. In Java, it appears that you are supposed to use a **File** object to determine whether a file exists, because if you open it as an **FileOutputStream** or **FileWriter** it will always get overwritten. By representing both files and directory paths, the **File** class also suggests poor design by violating the maxim "Don't try to do too much in a single class."

The IO stream library brings up mixed feelings. It does much of the job and it's portable. But if you don't already understand the decorator pattern, the design is non-intuitive, so there's extra overhead in learning and teaching it. It's also incomplete: there's no support for the kind of output formatting that almost every other language's IO package supports. (This was not remedied in Java 1.1, which missed the opportunity to change the library design completely, and instead added even more special cases and complexity.) The Java 1.1 changes to the IO library haven't been replacements, but rather additions, and it seems that the library designers couldn't quite get straight which features are deprecated and which are preferred, resulting in annoying deprecation messages that show up the contradictions in the library design.

However, once you *do* understand the decorator pattern and begin using the library in situations that require its flexibility, you can begin to benefit from this design, at which point its cost in extra lines of code may not bother you as much.

Exercises

1. Open a text file so that you can read the file one line at a time. Read each line as a **String** and place that **String** object into a **Vector**. Print out all of the lines in the **Vector** in reverse order.

2. Modify Exercise 1 so that the name of the file you read is provided as a command-line argument.

3. Modify Exercise 2 to also open a text file so you can write text into it. Write the lines in the **Vector**, along with line numbers, out to the file.

4. Modify Exercise 2 to force all the lines in the **Vector** to upper case and send the results to **System.out**.

5. Modify Exercise 2 to take additional arguments of words to find in the file. Print out any lines in which the words match.

6. In **Blips.java**, copy the file and rename it to **BlipCheck.java** and rename the class **Blip2** to **BlipCheck** (making it **public** in the process). Remove the //! marks in the file and execute the program including the offending lines. Next, comment out the default constructor for **BlipCheck**. Run it and explain why it works.